New Directions for Network Verification

Aurojit Panda[1], Katerina Argyraki[2], Mooly Sagiv[3], Michael Schapira[4], and Scott Shenker[5]

1 **UC Berkeley**
2 **EPFL**
3 **Tel Aviv University**
4 **Hebrew University of Jerusalem**
5 **UC Berkeley and ICSI**

1 Introduction

Verification — by which we mean the general practice of checking the correctness of a computer-based system before it is put into use — was first developed to check the correctness of hardware, and is now increasingly used in the software development process. While networks have been around for many decades and are now an essential piece of our computational infrastructure, only recently has verification been applied to ensure their correctness.[1] As a result, there is now a growing literature on systems that can verify that the current or proposed network configuration (as represented by router forwarding tables) obey various important invariants (such as no routing loops or deadends). These systems — which allow network operators to verify that their networks will operate correctly, in terms of some well-defined invariants – represent a valuable, and long overdue, step forward for networking, which for too long was satisfied with not only *best-effort* service but also *best-guess* configuration. In this position paper we critically review this recent progress, propose some next steps towards a more complete form of network verification, and end with a discussion of the formal questions posed to this community by the network verification agenda.

In the rest of this section, we provide some necessary background on networks and the current verification techniques. The application of verification to networking coincided with the rise of Software-Defined Networking (SDN). While not essential for the use of verification in networks, SDN provides a useful platform on which to deploy these tools so we discuss verification within the SDN context. Networks are comprised of two planes: the *data plane* which decides how packets are handled locally by each router (based on the local forwarding state and other information, such as state generated by previous packets), and the *control plane* which is a global process that computes and updates the local forwarding state in each router. In legacy networks, both planes are implemented in routers (with the data plane being the forwarding code or datapath, and the control plane being the global routing algorithm), but in SDN there is a clean separation between the two planes. The SDN control plane is logically centralized, and implemented in a few servers (called controllers) that compute and then install the necessary forwarding state. SDN-controlled routers only implement the data plane, executing a very simple datapath (OpenFlow [12]) in which the routing state is a set of $\langle match, action \rangle$ flow entries: all packets with headers matching the *match* entry are subject to the specified *action* which is often either to forward out a specific port (perhaps with a slightly modified header) or to drop the packet. One can think of this network state as the *configuration* of a router.

[1] There has been much work on verifying network protocols and their implementations, but until recently almost none on verifying a given network configuration.

The first wave of verification tools [7–10] analyzed the global behavior of a network made up of switches obeying this simple forwarding model. As a packet travels through the network, its next-hop is dictated by the routing state in the current router; thus, this network-wide behavior can be thought of as the composition of the routing state in each router. These early verification tools would take a snapshot of network state (either that which is already in the network, or that which the control is poised to insert into the network) and then verify whether some basic invariants held. These invariants (which are specified by the network operator) are typically quite simple and few in number: reachability (e.g., packets from host A can reach host B), isolation (e.g., packets from host A cannot reach host B), loop-freedom (no packet enters into an infinite loop), and no dead-ends (no packet arrives at a router which cannot forward it to another router or to the end-destination). Subsequent network verification tools (e.g., [1, 4, 15]) make the same assumptions about the datapath, but generalize along various other dimensions.

All of these tools leverage the fact that the simple forwarding model renders the datapath *immutable*; by this we mean that the forwarding behavior does not change until the control plane explicitly alters the routing state (which happens on relatively slow time scales). Thus, one can verify the invariants before each control-plane-initiated change and know that the network will always enforce the operator-specified invariants.

While the notion of an immutable datapath supported by an assemblage of routers makes verification tractable, it does not reflect reality. Modern enterprise networks are comprised of roughly 2/3 routers[2] and 1/3 *middleboxes* [19]. Middleboxes – such as Firewalls, WAN Optimizers, Transcoders, Proxies, Load Balancers, Intrusion Detection Systems (IDS) and the like – are the most common way to insert new functionality in the network datapath, and are commonly used to improve network performance and security.[3]

Just as the configuration of router is the state it uses to make forwarding decisions, the configuration of a middlebox is its set of policies (e.g., drop all Skype packets). The configuration of a network is the configurations of all of its routers, all of its middleboxes, and the topology of the physical network connecting these elements. The goal of verification is to ensure that a given network configuration supports a given invariant.

While useful, middleboxes are a common source of errors in the network [18], with middlebox misconfigurations account for over 40% of all major incidents in networks. Thus, one cannot ignore middleboxes when verifying network configurations.

However, middleboxes do not adhere to the simple forwarding model in routers. Many middleboxes have a *mutable* datapath, in which the handling of a packet depends not just on immutable forwarding state, but also on the sequence of previously encountered packets (e.g., a firewall packets from a flow into a network, but only if it has previously seen an outgoing packet from that flow). This dependence on the packet histories renders the datapath quite mutable (changing on packet timescales). This prevents the use of current verification techniques, because the control over packet behaviors is no longer centralized in the control plane but is, in the most general case, dependent on the packet histories seen by every middlebox.

Thus, we must find a way to verify network behavior in the presence of middleboxes, which means finding verification techniques that can deal with mutable datapaths. Since

[2] In this paper we do not distinguish between routers and switches, since they obey similar forwarding models.

[3] We should note that the NFV movement is moving middleboxes out of separate physical machines and into VMs that can be hosted on a cluster of servers; however, nothing in the move from physical to virtual middleboxes changes our story.

the forwarding behavior of a mutable datapath can depend arbitrarily on the past packet history, middleboxes render the network Turing-complete. Thus, any verification technique that copes with middleboxes will look much more like general program verification than the current generation of network verification tools. There are two main technical challenges to building the next generation of network verification tools:

- How do we model these mutable datapaths so that the complexity of verification is tractable?
- How can we feasibly analyze a network made up of these mutable datapaths?

We address these two challenges in the following sections, and then discuss how to formalize this approach and end by describing a set of open questions.

2 How to Model Middleboxes?

The natural approach for verifying mutable datapaths would be to apply standard program verification techniques to the code in each middlebox (and then somehow extend this to the network as a whole, which is the problem we address in the next section). The practical problem with this approach is that middlebox code is typically proprietary, and any approach that relies on middlebox vendors releasing their code is doomed to fail. Moreover, there is a deeper conceptual problem with this approach. The invariants specified by network operators often use abstractions, such as user identity, host identity, application-type (of the traffic), and whether or not the traffic is "suspicious" (e.g., after deep packet inspection). In fact, recent efforts to build policy languages are built around a similar set of abstractions.

The correctness of these abstractions often *cannot* be fundamentally verified (e.g., a middlebox in the middle of the network cannot always know for sure which host the packet came from given the various forms of spoofing or relaying available) or even precisely defined (e.g., what is suspicious traffic?). Yet these abstractions are quite useful (and already widely used) in practice, and operators are willing to live with their approximations (e.g., various techniques can be used to limit spoofing so that in some contexts host identification can use IP or MAC addresses as the basis for identification without great risk).

To allow reasoning in terms of high level abstractions without worrying about the various approximations that go into their definition, we model middleboxes in two parts: a reasonably simple abstract model that captures the action of a middlebox in terms of high-level primitives and an Oracle that is described by the set of abstractions it supports. The Oracle maps packets to one or more abstractions (e.g., this packet is from a Skype flow from host A and user X to host B and user Y), while the abstract model describes how the middlebox uses these abstractions (e.g., a middlebox might be configured to drop all suspicious packets, or only allow packets from host A to reach host B but no other hosts). For instance, for an IDS that identifies suspicious packets and forwards them to a scrubbing box, the Oracle part of the model is what determines which packets are suspicious and the abstract model is what dictates that such packets are forwarded to a scrubbing box.

The Oracles in different middleboxes may use very different techniques to map packets to abstractions. While operators care about the quality of this mapping, the goal of our network verification approach is to check that a network configuration correctly enforces invariants *assuming that the Oracles are correct*. Also, different Oracles may support different sets of abstractions (e.g., some firewalls may be able to identify Skype traffic, and others not), and this would be described as part of the middlebox model.

In constrast, the abstract models are fairly generic in the sense that the abstract model of a firewall applies to most firewalls. The degree of detail in these abstract models depend, to some degree, on the kind of invariants one wants to enforce. The basic network invariants of reachability and isolation only require that the abstract model describe the forwarding behavior (e.g., if and where each packet is forwarded). These basic invariants of reachability and isolation are our initial target, as this is by far the most important use of network verification. If one wants to support performance-oriented invariants then the abstract model must include timing information (e.g., what packet delays might occur). This is beyond the scope of our current efforts.

The separation of middlebox models into an abstract model and an Oracle has several advantages.

- It captures the fact that there are a limited number of middlebox "types", with many implementations of each. The abstract model applies to all of these implementations (and is fairly simple in nature), while the Oracle typically differs between implementations. Thus, our verification approach, which asks whether invariants are enforced assuming the Oracles are right, can be applied independent of the implementations.
- It differentiates between improvements in the Oracle (e.g., finding better methods for identifying application traffic), which is what consumes the bulk of the development effort, and verifying correctness of the network configuration.
- It could change the vendor ecosystem by allowing (or requiring) vendors to provide the abstract model (and a description of which abstractions their Oracle supports) along with their middlebox. Network operators could then perform verification, while vendors could keep their implementations private.
- While it would be useful for vendors to *verify* that their code obeys the abstract model (using standard code verification methods), what makes this approach particularly appealing is that vendors and operators alike can *enforce* that the middleboxes obey the abstract model. The abstract models (and we have built several of them) are so simple that they execute much faster than the actual middlebox implementation, so one can run these abstract models in parallel and ensure that the middleboxes take no action that does not obey the abstract model.

  Configuration: what does the abstract model

  Placement:

3 Network-Wide Verification

Modeling of individual middleboxes is only the first step; our ultimate goal is to veryify network-wide invariants in a network containing middleboxes and routers. There are numerous technical challenges (such as how to deal with loops), but in this short position paper we focus on the most challenging one: how can we feasibly perform verification in very large networks (containing on the order of thousands of middleboxes and routers)? To see why this is hard, consider the case of an isolation invariant (packets from host A cannot reach host B) in a network with many middleboxes. Since middlebox behavior depends not just on their configuration but on the packet history they've seen (since their datapath is mutable), verifying that this invariant holds, even if we ignore the possibility of packet loops, involves checking that there is no sequence of packets — involving packets sent from anywhere in the network — which includes a packet from host A reaching host B.

Without additional assumptions, this is not feasible. However, we note that the most common classes of middleboxes have an important property: the handling of a packet from host A to host B depends only on the sequence of packets (seen by the middlebox) between hosts A and B. That is, many mutable datapaths exhibit a useful kind of locality. For instance, in IP firewalls, the middlebox tracks established connections, and allows packets to pass if they are either explicitly permitted by policy or belong to an established connection. A connection between host A and B can only be established by host A and B. We therefore do not need to consider the actions of any other hosts in the network. We call such middleboxes RONO (Rest-Of-Network Oblivious) middleboxes. RONO is not a rare property; in fact, most middleboxes (including firewalls, WAN optimizers, load balancers, and others) can be configured to be RONO (e.g., caches are RONO if they support ACLS). Moreover, one can verify whether a middlebox is RONO by statically analyzing its abstract model.

Note that the composition of RONO middleboxes is also RONO. As a result, in a network containing only RONO middleboxes we can verify reachability properties on a small subset of the network (the path between two hosts) and these properties would equivalently hold in the context of the wider network. We have leveraged this fact to verify correctness of a 30,000 middlebox network in under two minutes.

4 Common Myths about Networks and SDN Verification

We next highlight some common myths about SDN networks and contrast them with the reality of today's networks.

Myth #1: *SDN networks only have controllers and OpenFlow routers, with all complicated (particularly mutable) packet processing done at the controller (e.g., learning switches, firewalls).* The early SDN literature [3, 14] relied on examples where anything expressible using OpenFlow rules was pushed down to routers, while anything more complex was implemented in the controller. Complicated functionality included both complex processing or transcoding that could not be performed at routers and simple tasks that required mutability (*e.g.,* learning switches and stateful firewalls). However, doing this processing at the controller does not scale and, in reality, middleboxes are used to provide most of the processing functions not implementable on switches, and most routers provide some mutable behavior (e.g., learning switch).

Myth #2: *Centralization, as provided by SDN, is what makes current network verification efforts possible.* Centralization is neither necessary nor sufficient for network verification. Not necessary: Verification in a network with immutable datapaths only requires being able to access router forwarding state, and current commercial network verification efforts target can do this in legacy networks by using commonly available commands to read this forwarding state. Not sufficient: Regardless of SDN, current network verification efforts cannot verify networks which have middleboxes with mutable datapaths (which describes almost all real networks).

Myth #3: *Middleboxes are an aberration that will be eliminated by the rise of SDN.* Quite the opposite is true. Not only are middleboxes here to stay – as the NFV movement promoting their move to software has gotten significant commercial traction (comparable to or exceeding that of SDN) and configuration efforts like Congress [16] are treating them as first-class network citizens – but SDN itself has been evolving to incorporate middleboxes [**?**].

Myth #4: *We should write all network code and configuration in declarative languages, because their use makes verification easy.* While quite useful when dealing with immutable datapaths, declarative languages lose much of their charm in the presence of mutable state and thus are not ideally suited for writing middleboxes. Decidability is a tricky issue for verification. Intuitively, the cost of verification is proportional to the complexity of the program and the required invariant. Even without side-effects there are only very few cases in which datalog verification is decidable [5]. The situation gets worth when simple side-effects are introduced [11]. Declarative languages are also commonly proposed as a mechanism for specifying network configuration [16]. While this use might make configuration easier it should not be considered a panacea for verification. The decidability problem when analyzing mutable datapaths is primarily a result of the dataplane being Turing complete, not configuration complexity.

Once one discards these myths, it becomes clear that network verification efforts must directly confront the presence of mutable datapaths. While the approach described here may not be optimal, it is currently the only one that confronts the reality of today's and tomorrow's networks. It is time for taking the next step in network verification, and in the next section we outline some of the challenges that this task poses for our community.

5 Formalizing the Mutable Data Plane

The previous sections argued why a new approach to network verification is needed and briefly outlined what it might look like. **[Here we present some early...]**

5.1 Verification Problem

Prior work [2] has shown that precisely specifying the desired behavior of a program (or network) is hard. The lack of a precise specification is one of the biggest challenges when performing program and network verification. In this section we limit our verification efforts to invariants from a commonly a universal, important subset, Reachability, and develop formalism to reason about these properties.

5.1.1 Network Reachability

The primary function of networks is to allow hosts to communicate with each other. Reachabality, the property that a certain class of packets sent from host A can reach host B, and its converse, isolation, are fundamental to networks; all useful networks must satisfy some set of reachability properties and their verification is thus universally important.

To formally express both reachability and isolation in temporal logic, we define two relations: $\mathit{Send}(n, p)$ indicating some network entity (node) n sent a packet p (at some time), and $\mathit{Recv}(n, p)$ indicating node n received packet p. Given these relations any reachability property can be expressed in CTL as

$$\forall p \in \text{Packet} : \mathit{Send}(\mathit{src}, p) \land \mathit{Predicate}(p) \implies \Diamond(\mathit{Recv}(\mathit{dest}, p))$$

This temporal logic statement says if a packet p sent by src meets the predicate $\mathit{Predicate}$, then the packet is eventually received at dest. Similarly, isolation can be formally expressed as

$$\forall p \in \text{Packet} : \mathit{Send}(\mathit{src}, p) \land \mathit{Predicate}(p) \implies \Box(\neg \mathit{Recv}(\mathit{dest}, p))$$

Predicate in the definition above is specified using the same abstractions used to specify network policies, *i.e.,* either in terms of packet header fields (source, destination, etc.) or in terms of the abstraction in §2. These reachability properties are thus defined in terms of the abstractions implemented by the Oracle for one or more middleboxes in the network. For example, a property saying no SSH traffic can reach a server d can be expressed as

$$\forall s \in \text{Node} : \mathit{Send}(s, p) \land \mathit{ssh}(p) \implies \Box(\neg \mathit{Recv}(d, p)) \tag{1}$$

Listing 1 Model for an IDS

```
1 oracle suspicious? (packet: Packet) : Boolean;
2
3
4 model ids (n: Node, p: Packet) = {
5   when suspicious?(p) =>
6       forward {}
7   default =>
8       forward {p}
9 }
```

where $ssh(p)$ returns true if an Oracle classifies the packet as belonging to an SSH connection. We reason about reachability and isolation properties assuming the Oracles are correct. Verifying the isolation property in Equation 1 therefore requires answering the question: "assuming SSH traffic is correctly identified, can a packet belonging to an SSH connection reach d?"

5.2 Formalizing Middlebox Semantics

[Panda: I think we should cut this bit, or move it elsewhere] Along with reachability properties, verification also requires that we specify middleboxes. We expect programmers or network operators will specify middlebox models (which include both an oracle and a generic model) using a constrained programming language. Listing 1 shows an example of such a specification for an intrusion detecion system (IDS). The IDS oracle provides one abstraction, `suspicious?`, defined in the first line. The abstract model is defined in Lines 4 – 9 and uses the abstraction. First we check to see if the packet is suspicious (the Oracle's decision here might be based on what packets it has seen previously), and drop the packet (Line 6) or forward the packet (Line 9) depending on the value returned by the Oracle. *[Panda: End of where we would cut until]*

Next we develop abstract semantics for middleboxes. We use this to reason about more general properties (such as composition) that apply to all middleboxes. Our semantic model is defined over a potentially infinite set of packets P. All packets in P are augmented to include information about their location (*i.e.,* the middlebox or switch port), and can include additional fields such as the data origin, target-host, etc. We also define the operator $\doteq$, where $p_1 \doteq p_2$ implies that packets p_1 and p_2 are identical except for their location. Finally, we use P^* to represent the set of all (potentially unbounded) sequence of packets.

We define a middlebox m, as a function $m\colon P \times P^* \to 2^P$ which takes a packet ($p \in P$) and a history ($h \in P^*$) of all the packets that have previously been processed by m and produces a (possibly empty) set of packets $m(p, h)$. Given this model a switch is a simple function for which $m(p, h) \doteq \{p\}$. Similarly a simple firewall, f (whose decision process is represented by *allowed*) can be expressed as

$$f(p,h) = \begin{cases} \{p'\} \;\; p' \doteq p & \text{if } \mathit{allowed}(p,h) \\ \{\} & \mathit{otherwise} \end{cases}$$

5.3 Composing Middleboxes

Ideally, we would like to be able to reason about the network *compositionally*, *i.e.,* the correctness of the network should follow from the correctness of smaller, simpler components.

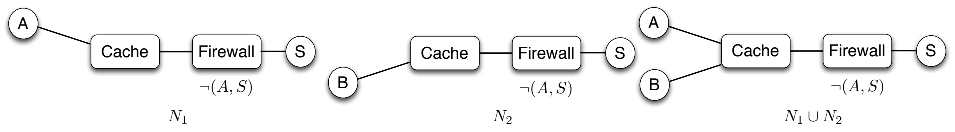

Figure 1 Example where networks are not composable with respect to reachability properties.

Compositional reasoning can reduce the cost of verification and enables incremental verification in response to changes in the network. Compositionality has been important for making verification tractable in other domains, for instance the use of rely-gurantees [6,13] has been very important for verifying concurrent programs.

We start by defining what it means for us to be able to compositionally verify a network. Define the union of two networks N_1 and N_2 in the natural way, *i.e.,* $N_1 \cup N_2$ contains the union of all nodes and links in each of N_1 and N_2. Consider two networks: N_1 where property P_1 holds (represented as $N_1 \models P_1$) and N_2 where property P_2 holds. We can compose the proofs for properties P_1 and P_2 if and only if both P_1 and P_2 hold for $N_1 \cup N_2$. More formally, we can verify properties P_1 and P_2 compositionally for network N if for any $N_1 \subset N$ and $N_2 \subset N$

$$\frac{N_1 \models P_1, N_2 \models P_2}{(N_1 \cup N_2) \models P_1 \wedge P_2}$$

Generally, one cannot perform compositional verification of reachability properties. As an example of a case where two network cannot, consider N_1 and N_2 in Figure 1. The cache in this case records responses from S for previous requests. On receiving a request, if the cache has a saved response it returns that directly, otherwise it forwards the request, unmodified, to the firewall. The firewall drops all requests sent from A to S, but otherwise forwards all other requests and responses unmodified. In network N_1, A can never receive a response from S (thus is isolated). However in the composed network $N_1 \cup N_2$, if B sends a request r and receives response r' from S, then A can also request r and receive r'.

5.3.1 Rest of Network Oblivious Middleboxes

There are an important subset of networks where compositional verification can be used even for reachability properties. These networks only contain Rest-Of Network Oblivious middleboxes (§3), middleboxes whose behavior for a pair of hosts depends only on the traffic sent between these hosts. Define restriction $h|_{(A,B)}$ for history $h \in P^*$ to be the subsequence of h containing only those packets that were sent between host A and B. We define a middlebox m to be RONO if and only if

$$f(p,h) = f(p,h|_{(A,B)}) \forall p.src = A \wedge p.dest = B \text{and}$$
$$f(p,h) = f(p,h|_{(A,B)}) \forall p.src = B \wedge p.dest = A$$

Not all compositions of RONO middleboxes are themselves RONO. In [17] we identify conditions where a network of RONO middleboxes supports compositional verification.

6 Open Problems for Network Verification

6.1 Decidability of Verification

Our abstract models for middleboxes such Firewall an IDSs have infinite state. Therefore, in general it may be undecidable to check properties like network isolation. We are currently working on a programming language for specifying abstract middleboxes, which allows to algorithmically check interesting network properties, including network isolation.

6.2 Specification

The biggest problem of program (and network) verification, as already observed in [2], is the absence of exact specification of what the system intends to do. For example, we do not know how to exactly formalize the correctness of middleboxes such as IDSs. We have argued that network isolation is a useful property. It may be interesting to develop methods for defining properties of networks in terms of the Oracles.

6.3 Correctness Preserving Transformations and Parametric Topologies

It may be possible to show that certain correctness properties are preserved by the addition of certain middleboxes in certain ways. For example, we may want show that the insertion of certain kinds of middleboxes e.g., WAN cannot lead to new violations of network isolation. This will be very nice as it will give principles for changing network topologies. Another open question is can one verify a parametric network of middleboxes? For example, the Azure network seems to be built out many parallel paths for scalability. Can we verify one path and then show that there are no interaction?

[Maybe add the ADT bit?]

References

1 Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. NetKAT: Semantic foundations for networks. POPL, 2014.

2 Richard A. DeMillo, Richard J. Lipton, and Alan J. Perlis. Social processes and proofs of theorems and programs. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pages 206–214, 1977.

3 Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick McKeown, and Scott Shenker. Nox: towards an operating system for networks. *ACM SIGCOMM Computer Communication Review*, 38(3):105–110, 2008.

4 Arjun Guha, Mark Reitblatt, and Nate Foster. Machine-verified network controllers. PLDI, 2013.

5 Alon Y. Halevy, Inderpal Singh Mumick, Yehoshua Sagiv, and Oded Shmueli. Static analysis in datalog extensions. *J. ACM*, 48(5):971–1012, September 2001.

6 Cliff B. Jones. Specification and design of (parallel) programs. In *IFIP Congress*, pages 321–332, 1983.

7 Peyman Kazemian, Michael Chang, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. Real time network policy checking using header space analysis. NSDI, 2013.

8 Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. NSDI, 2012.

9 Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. Veriflow: Verifying network-wide invariants in real time. NSDI, 2013.

10 Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, Brighten Godfrey, and Samuel Talmadge King. Debugging the Data Plane with Anteater. SIGCOMM, 2011.

11 Johann A. Makowsky, J.-C. Gregoire, and Shmuel Sagiv. The expressive power of side effects in prolog. *J. Log. Program.*, 12(1&2):179–188, 1992.

12 Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling Innovation in Campus Networks. SIGCOMM CCR '08.

13 Jayadev Misra and K. Mani Chandy. Proofs of networks of processes. *IEEE Trans. Software Eng.*, 7(4):417–426, 1981.

14 Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, David Walker, et al. Composing software defined networks. NSDI, 2013.

15 Tim Nelson, Andrew D. Ferguson, Michael J. G. Scheer, and Shriram Krishnamurthi. A balance of power: Expressive, analyzable controller programming. NSDI, 2014.

16 OpenStack. Congress. `https://wiki.openstack.org/wiki/Congress` retrieved 01/08/2015.

17 Aurojit Panda, Ori Lahav, Katerina J. Argyraki, Mooly Sagiv, and Scott Shenker. Verifying isolation properties in the presence of middleboxes. *CoRR*, abs/1409.7687, 2014.

18 Rahul Potharaju and Navendu Jain. Demystifying the dark side of the middle: a field study of middlebox failures in datacenters. In *IMC*, 2013.

19 Justine Sherry, Shaddi Hasan, Colin Scott, Arvind Krishnamurthy, Sylvia Ratnasamy, and Vyas Sekar. Making middleboxes someone else's problem: Network processing as a cloud service. SIGCOMM, 2012.